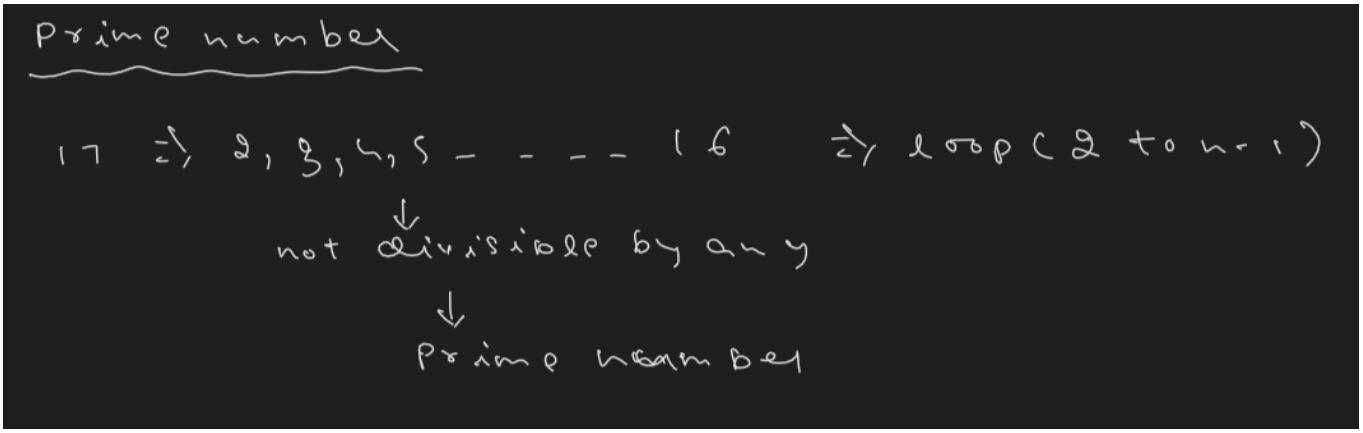
🎯 Count Primes

https://leetcode.com/problems/count-primes/




```java
class Solution {
    public int countPrimes(int n) {
        int count = 0;
        for (int i = 2; i < n; i++) {
            if (isPrime(i)) {
                count++;
            }
        }
        return count;
    }

    public boolean isPrime(int x) {
        for (int d = 2; d <= x-1; d++) {
            if (x % d == 0) {
                return false;
            }
        }
        return true;
    }
}
```

The brute-force solution we implemented above results in a **Time Limit Exceeded (TLE)** error. This happens because, in the worst-case scenario, its time complexity is **O(n²)**.

Now, consider the problem constraints:
The maximum value of n can be up to 10^6.
So, in the worst case, the number of operations would be on the order of 10^{12} (i.e., $10^6 \times 10^6$).

We know that a typical computer can execute about 10^8 **operations per second**. Since 10^{12} is far beyond that, this brute-force solution is simply too slow and will inevitably lead to a TLE error.

To check whether a number n is **prime**, I don't need to try dividing it by all numbers up to n.
Instead, I can reduce the number of checks to √**n (square root of n)**.

Why?

If n is **not** a prime number, then it can be expressed as the product of two integers:

$n = a \times b$

There are three possible cases:

1. **Case 1: a = b**
   In this case:
   $n=a\times a=a^2 \Rightarrow a=\sqrt{n}$
2. **Case 2: a > b**
   As a increases, b must decrease to maintain the equality $a\times b=n$.
   So b becomes smaller than $\sqrt{n}$.

3. **Case 3: a < b**
   Similarly, since a is smaller, it must be **less than** $\sqrt{n}$ to keep the product equal to n.

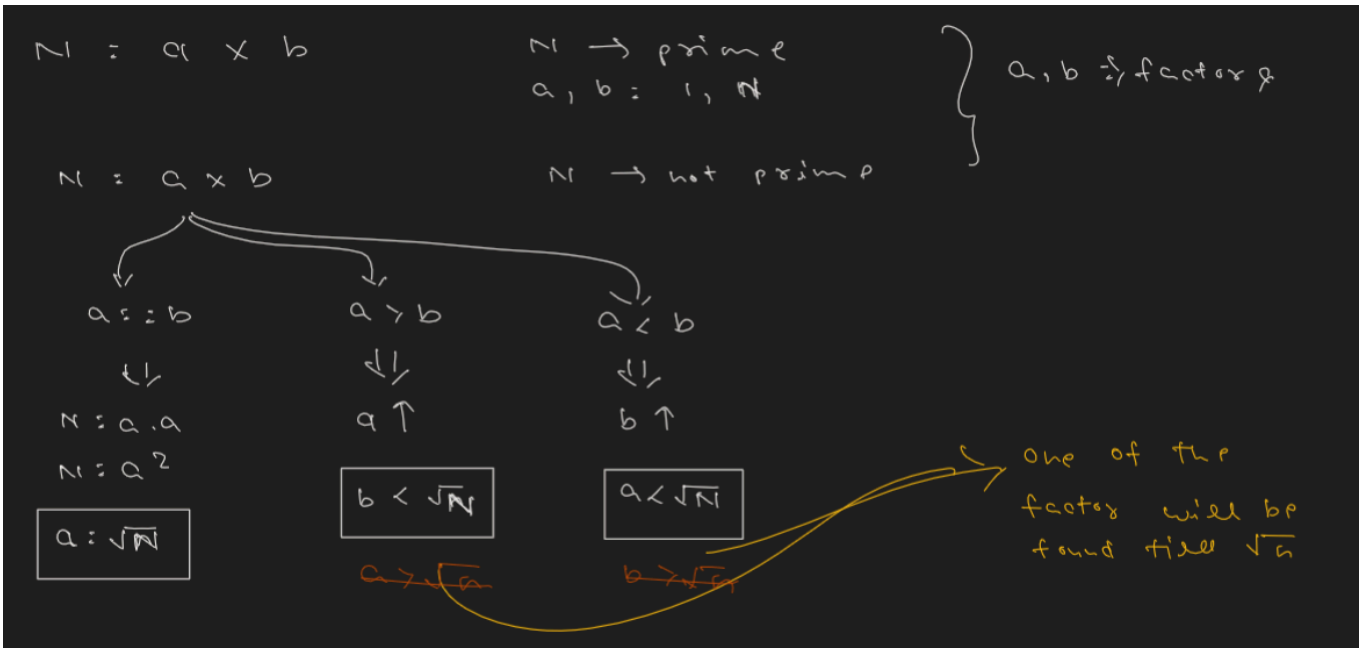
Final Conclusion:

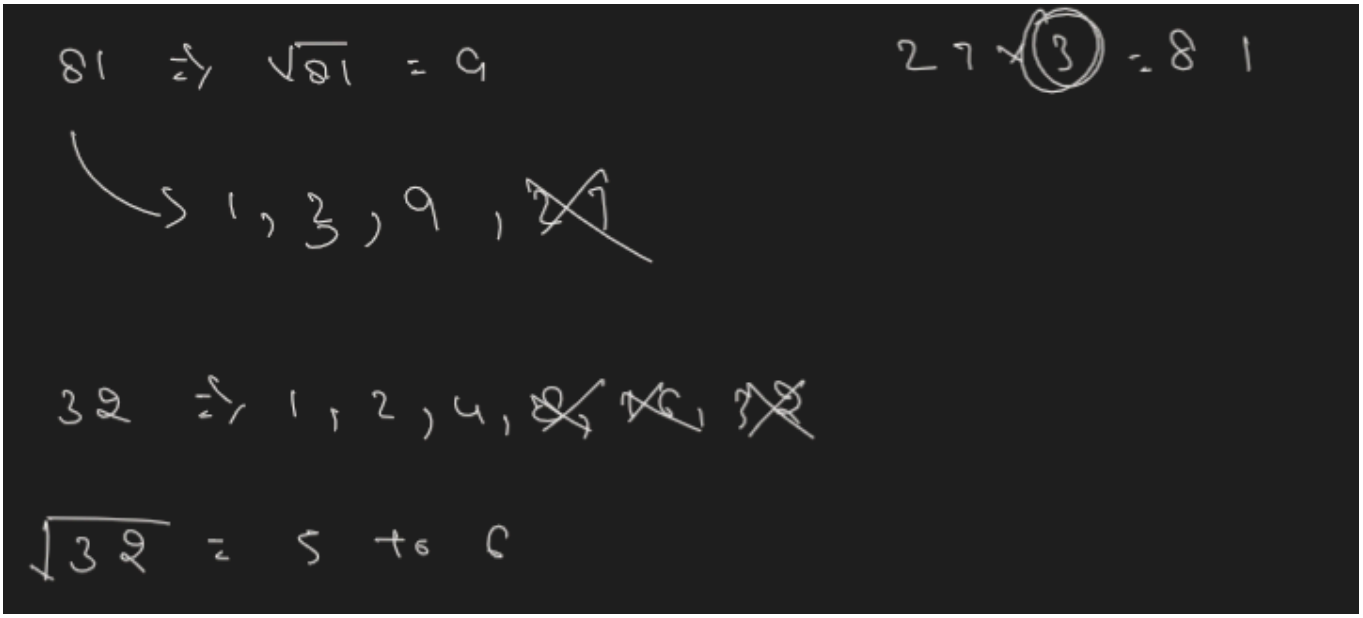
In **all three cases**, **at least one factor (either a or b) will be ≤ $\sqrt{n}$**.
So, if a number n has any factor other than 1 and itself, **you will find at least one such factor ≤ $\sqrt{n}$**.

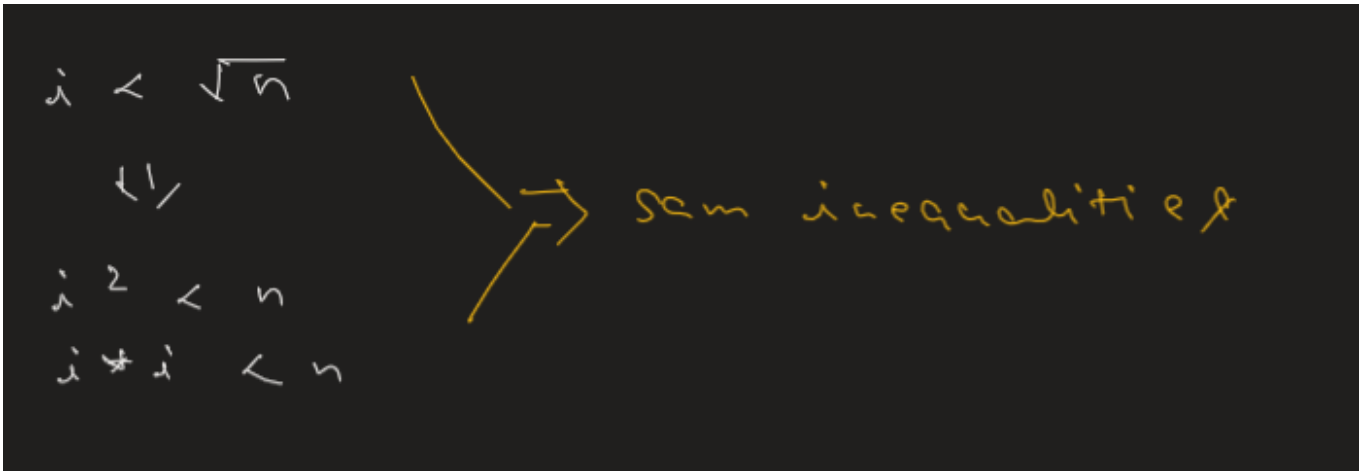
Ex: 36 => 2, 4

Therefore:

To check if n is prime, it's enough to try dividing it only by numbers from 2 to $\sqrt{n}$.
If none of them divide n, then n is **prime**.








```java
public boolean isPrime(int x) {
    if (x <= 1) return false;

    for (int d = 2; d * d <= x; d++) {
        if (x % d == 0) {
            return false;
        }
    }
    return true;
}
```

The smart brute-force solution we implemented above results in a **Time Limit Exceeded (TLE)** error. This happens because, in the worst-case scenario, its time complexity is **O(n*√n) ⇒ O($n^{3/2}$)**.

Now, consider the problem constraints:
The maximum value of `n` can be up to **10^6**.
So, in the worst case, the number of operations would be on the order of **10^9** (i.e., $10^{6X3/2}$).

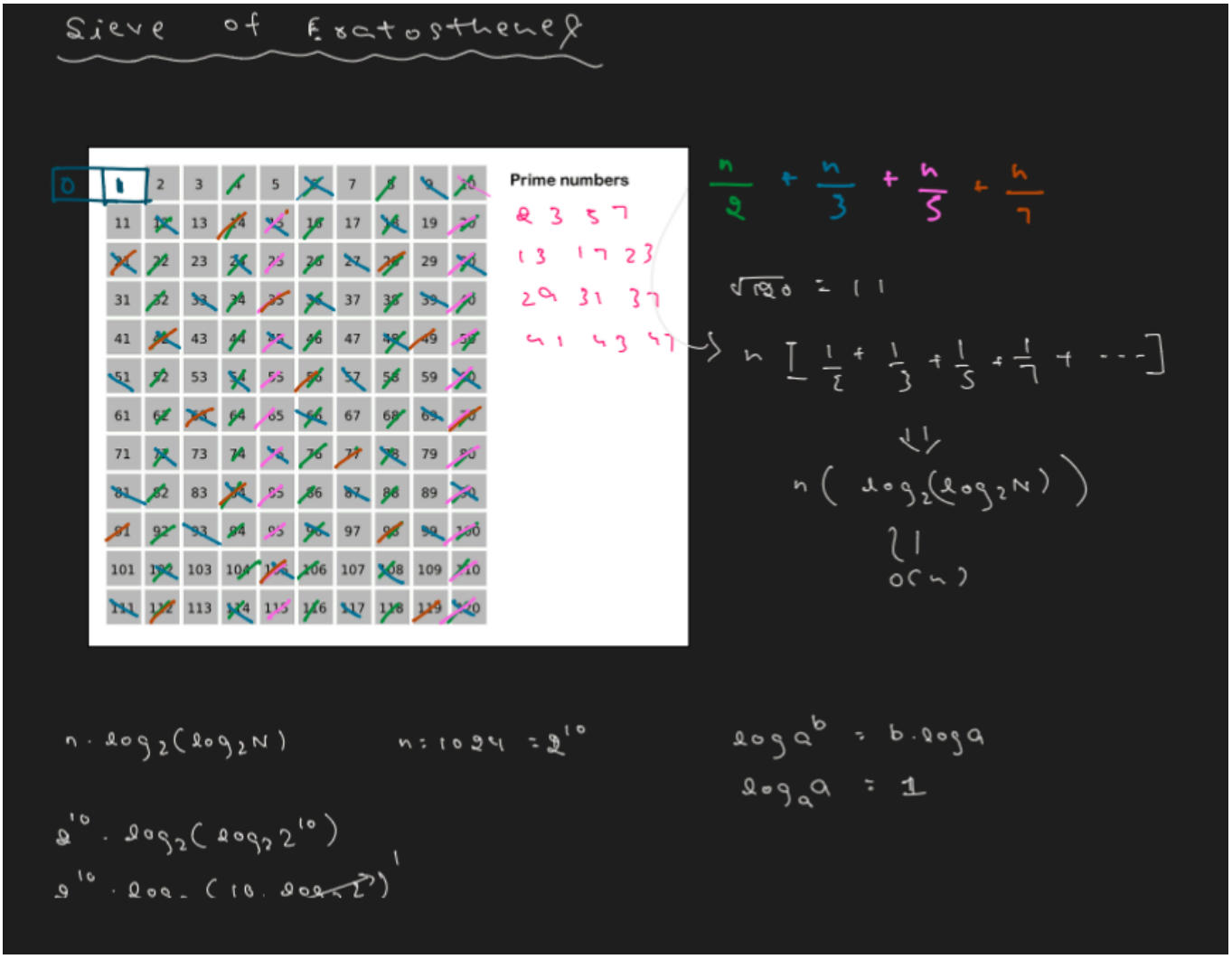
We know that a typical computer can execute about **10^8 operations per second**. Since **10^8** is far beyond that, this smart brute-force solution is simply too slow and will inevitably lead to a TLE error.

🧮 What is the Sieve of Eratosthenes?

The **Sieve of Eratosthenes** is an ancient and efficient algorithm used to **find all prime numbers up to a given number `n`**.

Instead of checking each number individually (like in brute-force or `isPrime()`), it uses a clever **marking technique** to eliminate non-primes in bulk — significantly reducing the time complexity.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120



🧠 How It Works:

1. **Create a boolean array**
   - Make an array `isPrime[]` of size `n` and set all entries to `true`.
   - Mark `0` and `1` as `false` since they are not prime.

2. **Start from 2**
   - Begin with the first prime number `p = 2`.

3. **Mark multiples as non-prime**
   - For each `p`, mark all multiples of `p` starting from `p*p` as `false` (i.e., not prime).
   - For example, for `p = 2`, mark `4, 6, 8, ...`
     For `p = 3`, mark `9, 12, 15, ...`, etc.

4. **Repeat until √n**
   - Only go up to `p * p < n`, because any composite number less than `n` will have a factor ≤ √n.

5. **Count the `true` values**
   - The remaining `true` values in `isPrime[]` are all primes.

🧮 Time & Space Complexity:

- **Time:** O(n log log n)
- **Space:** O(n)

Very efficient for values of n up to 10^6 or more.

```
public class Main {
	public static void main(String[] args) {
		int n = 100;
		System.out.println(primeSieve(n));
	}

	// 1 -> not prime | 0 -> prime
	public static int primeSieve(int n) {
		int[] prime = new int[n];
		prime[0] = prime[1] = 1; // 0 & 1 is not a prime number

		for (int i = 2; i*i < prime.length; i++) {
			if (prime[i] == 0) {
				for (int j = i*i; j < prime.length; j+=i) {
					prime[j] = 1;
				}
			}
		}

		int c = 0;
		for (int i = 2; i < prime.length; i++) {
			if (prime[i] == 0) {
				c++;
			}
		}
		return c;
	}

}
```

Note: Dry Run for n = 30.

```
class Solution {
    public int countPrimes(int n) {
		if(n <= 1){
		    return 0;
		}

		int[] prime = new int[n];
		prime[0] = prime[1] = 1; // 0 & 1 is not a prime number

		for (int i = 2; i*i < prime.length; i++) {
			if (prime[i] == 0) {
				for (int j = i*i; j < prime.length; j+=i) {
					prime[j] = 1;
				}
			}
		}

		int c = 0;
		for (int i = 2; i < prime.length; i++) {
```

```
                if (prime[i] == 0) {
                    c++;
                }
            }
            return c;
    }

}
```

🎯 Palindromic Substrings

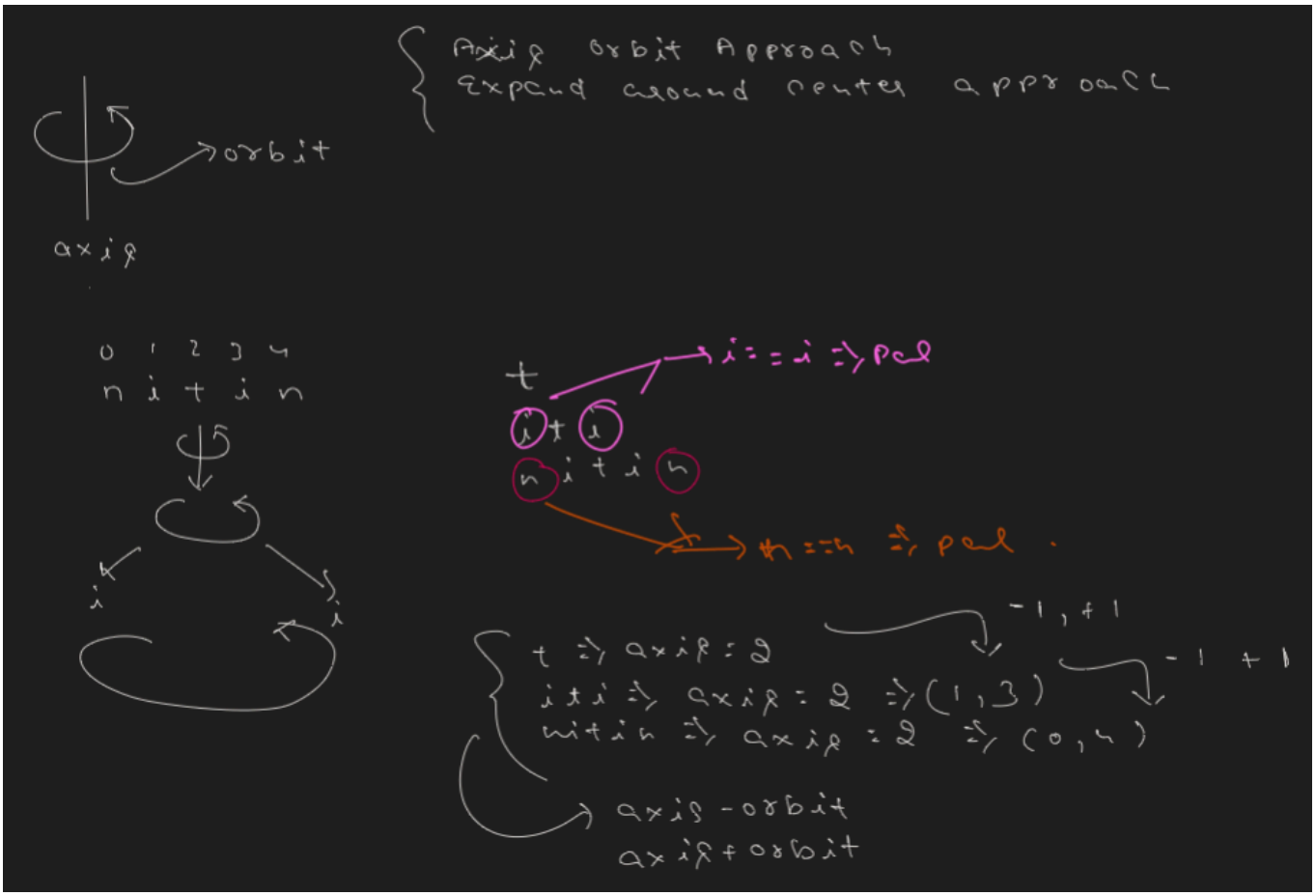
https://leetcode.com/problems/palindromic-substrings/

```
class Solution {
    public int countSubstrings(String s) {
        int c = 0;
        for(int i=0; i<s.length(); i++){
            String op = "";
            for(int j=i; j<s.length(); j++){
                op = op + s.charAt(j);
                // System.out.println(op);
                if(isPalindrome(op)){
                    c = c+1;
                }
            }
        }
        return c;
    }

    public static boolean isPalindrome(String x){
        int i = 0;
        int j = x.length()-1;

        while(i<j){
            if(x.charAt(i) != x.charAt(j)){
                return false;
            }
            i++;
            j--;
        }
        return true;
    }

}
```

Variations of the Palindromic Substring Problem:

1. Count all palindromic substrings
2. Find the longest palindromic substring
3. Count all odd-length palindromic substrings
4. Count all even-length palindromic substrings
5. Find the longest odd-length palindromic substring
6. Find the longest even-length palindromic substring

🔄 Expand Around Center/ Axis Orbit Approach



🧠 Intuition

Suppose we take the character `'t'` as the **axis** (i.e., the center) of a potential palindrome.

✅ Step 1: Start with Orbit = 0

- At `orbit = 0`, we are only looking at the center itself, which is `'t'`.
- A single character is always a palindrome.
- So, the substring `"t"` is a valid palindrome.

🔁 Step 2: Expand Outwards – Orbit = 1

- Now, we look in **both directions** from the center:
  - `axis - 1` and `axis + 1`
- Suppose both characters at these positions are `'i'` and `'i'`.
- Since both are the same, the substring `"iti"` is a palindrome too.

🔁 Step 3: Expand Further – Orbit = 2

- Next, we check again:
  - `axis - 2` and `axis + 2`
- Suppose we find `'n'` and `'n'` at these positions.
- Again, both are equal, so the full substring `"nitin"` is also a palindrome.

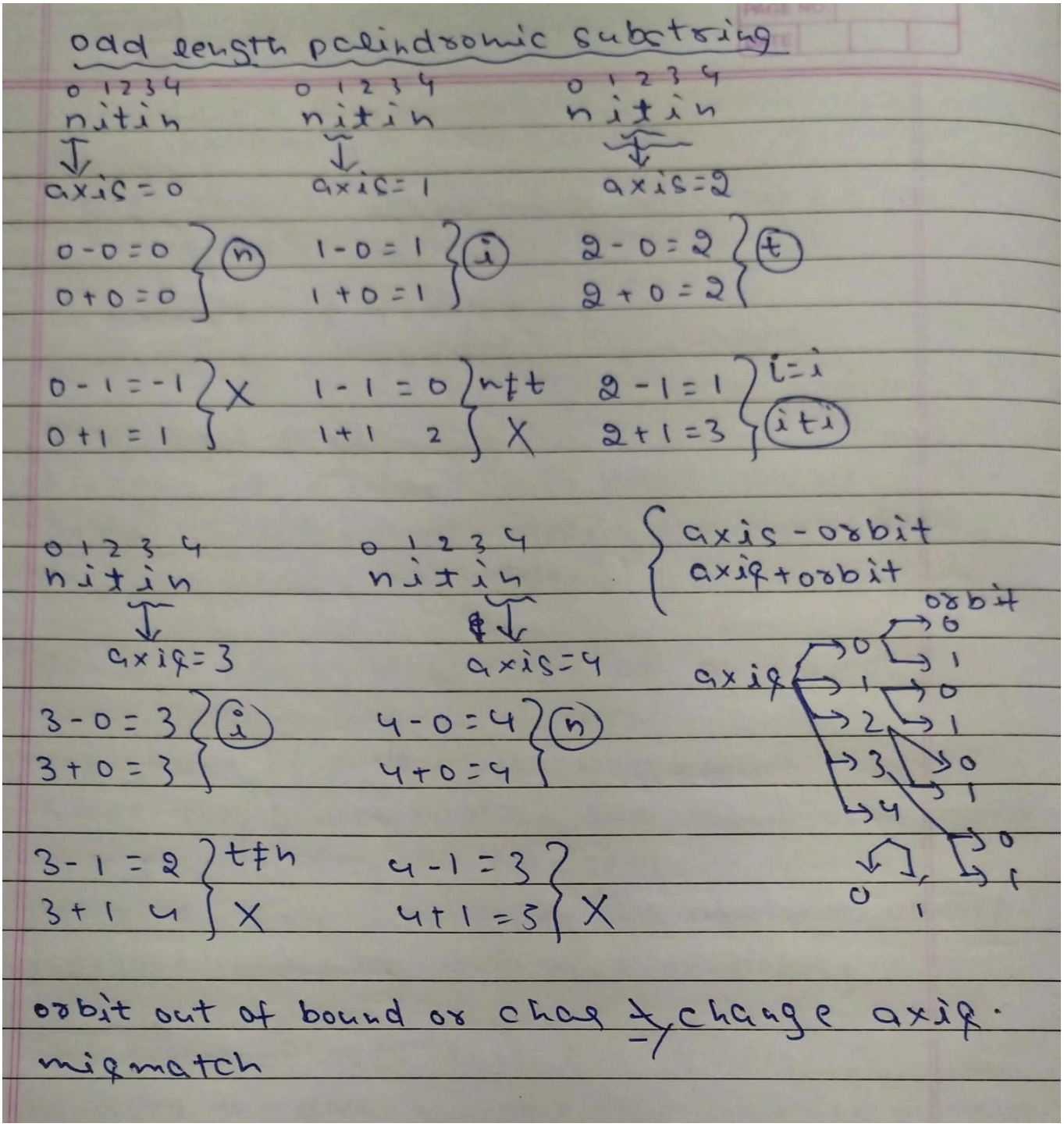
🧩 Summary of the Idea:

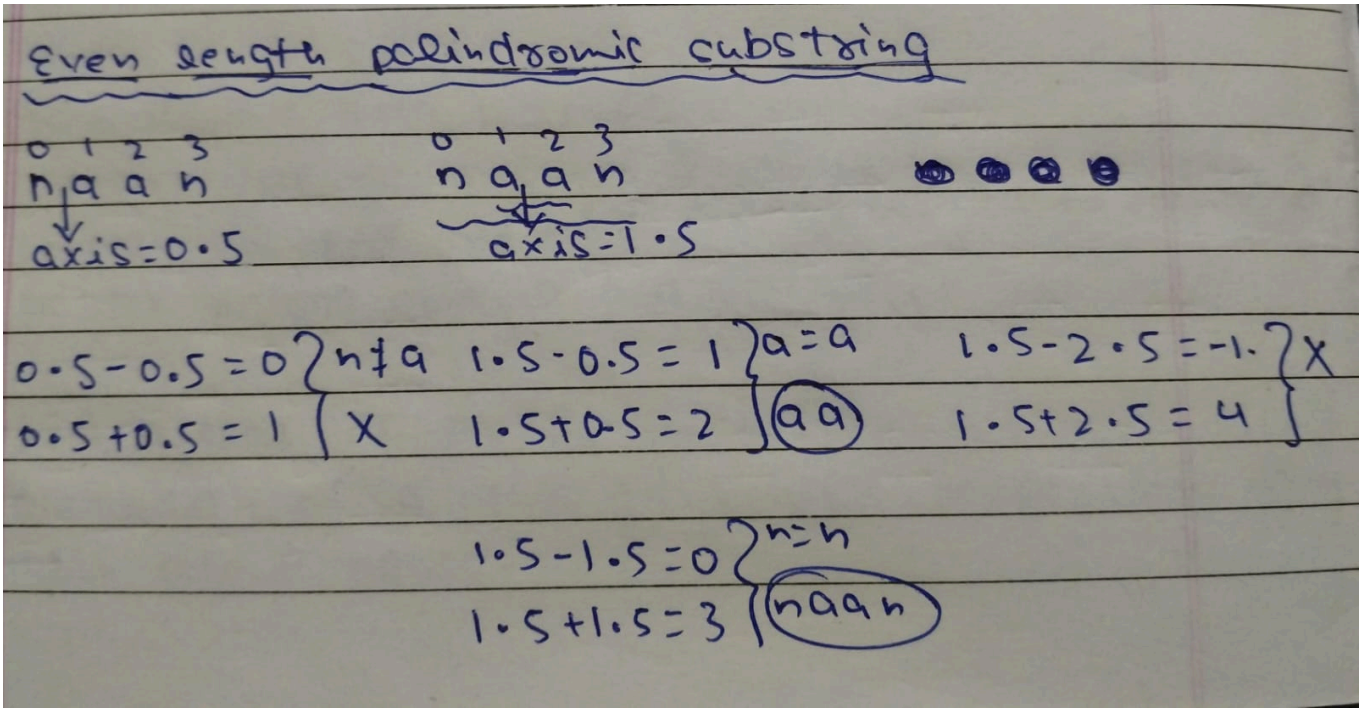
- We **start from a center** (the axis).
- We **expand symmetrically outward** (using orbit).
- At each step, we check if the characters on both sides are equal.

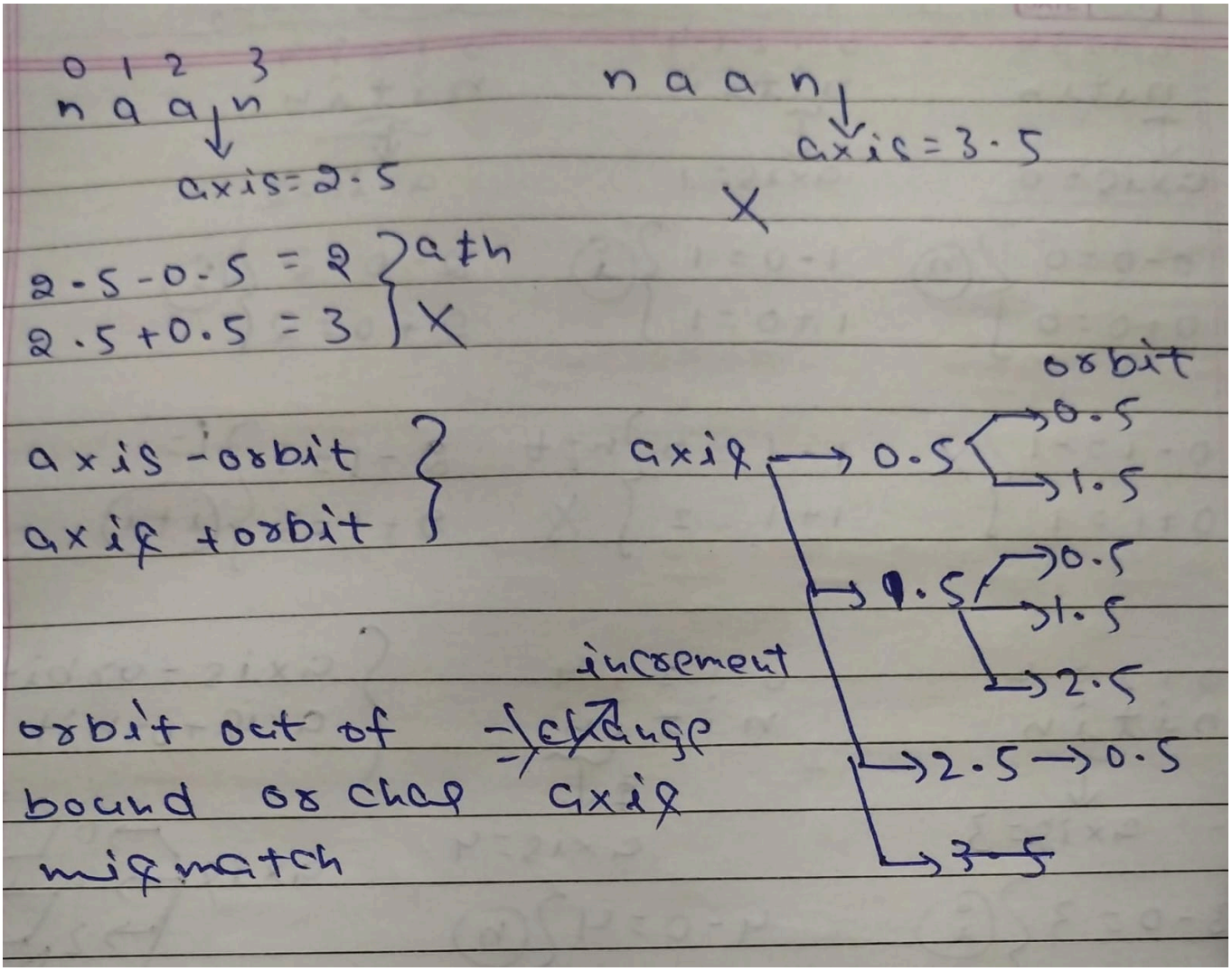
- If they are, we found another palindromic substring.
- This process continues until the characters don't match or we go out of bounds.

💡 **Key Insight:**

If a smaller substring is a palindrome and we **wrap matching characters around it**, the new substring is also a palindrome.








```java
public class Main {
	public static void main(String[] args) {
		String s = "naan";
		System.out.println(palindromicSubstringCount(s));
	}

	public static int palindromicSubstringCount(String s) {
```

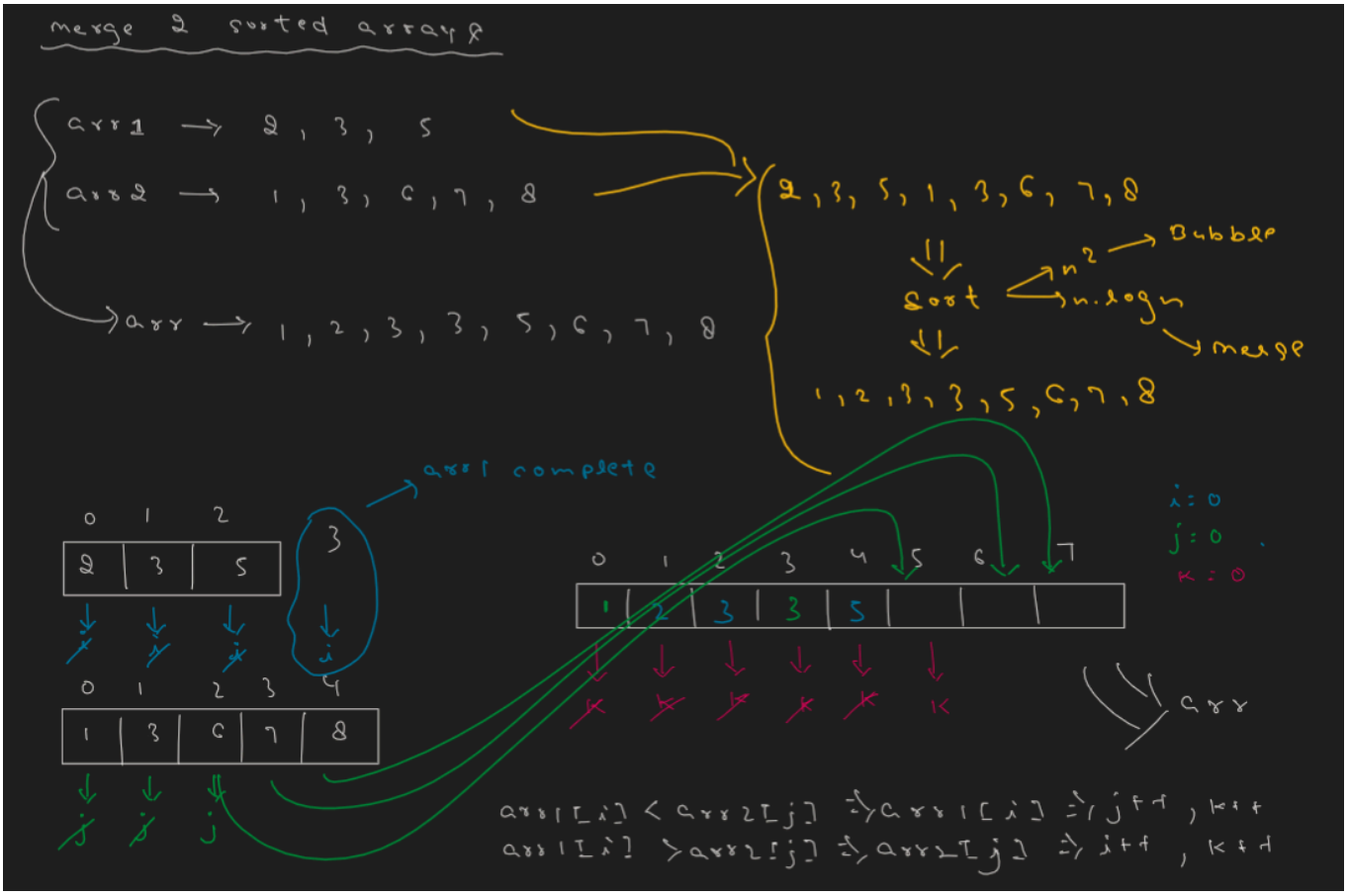
```
            // odd length substring → O(N²)
            int odd = 0;
            for (int axis = 0; axis < s.length(); axis++) {
                  for (int orbit = 0; axis - orbit >= 0 && axis + orbit <
s.length(); orbit++) {
                        if (s.charAt(axis - orbit) != s.charAt(axis + orbit)) {
                              break;
                        }
                        odd++;
                  }
            }

            // even length substring → O(N²)
            int even = 0;
            for (double axis = 0.5; axis < s.length(); axis++) {
                  for (double orbit = 0.5; axis - orbit >= 0 && axis + orbit <
s.length(); orbit++) {
                        if (s.charAt((int) (axis - orbit)) != s.charAt((int) (axis
+ orbit))) {
                              break;
                        }
                        even++;
                  }
            }
            return odd + even;
      }

}
```

```
class Solution {
    public int countSubstrings(String s) {
        return palindromicSubstringCount(s);
    }

    public static int palindromicSubstringCount(String s) {
        // odd length substring
        int odd = 0;
        for (int axis = 0; axis < s.length(); axis++) {
            for (int orbit = 0; axis - orbit >= 0 && axis + orbit < s.length(); orbit++) {
                if (s.charAt(axis - orbit) != s.charAt(axis + orbit)) {
                    break;
                }
                odd++;
            }
        }

        // even length substring
        int even = 0;
        for (double axis = 0.5; axis < s.length(); axis++) {
            for (double orbit = 0.5; axis - orbit >= 0 && axis + orbit < s.length();
orbit++) {
                if (s.charAt((int) (axis - orbit)) != s.charAt((int) (axis + orbit))) {
                    break;
                }
                even++;
```

```
            }
        }
        return odd + even;
    }

}
```

🎯 Merge Sorted Array

https://leetcode.com/problems/merge-sorted-array/




```
public class Main {
	public static void main(String[] args) {
		int[] arr1 = { 2, 3, 5 };
		int[] arr2 = { 1, 3, 6, 7, 8 };
		int[] arr = merge(arr1, arr2);
		for (int i = 0; i < arr.length; i++) {
			System.out.print(arr[i] + " ");
		}
	}

	public static int[] merge(int[] arr1, int[] arr2) {
	  int n = arr1.length;
	  int m = arr2.length;

	  int[] arr = new int[n+m];

	  int i = 0;
	  int j = 0;
```

```
        int k = 0;

        while(i<n && j<m){
            if(arr1[i] < arr2[j]){
                arr[k] = arr1[i];
                i++;
            }
            else{
                arr[k] = arr2[j];
                j++;
            }
            k++;
        }

        while(i<n){
            arr[k] = arr1[i];
            k++;
            i++;
        }
        while(j<m){
            arr[k] = arr2[j];
            k++;
            j++;
        }
        return arr;
      }

}
```